Winter 2019 - COMP 520 Compiler Design
Final Report

Group 07
Zhuocheng Du - 260673297
Anqi Li - 260654876
Zisheng Wang - 260619653

1 Introduction

Go is a statically typed, complied general-purpose language developed by Google. It has garbage collection like Java, gives high priority to concurrency, and supports multiple programming paradigms. It is well designed for "multicore processors, cloud services, computation clusters, and networked systems"[1].

In this project, we have implemented a compiler for GoLite programming language, which is a strict subset of Go programming language. Some features of Go have been eliminated for the purpose of the course. For example, slice literals are not supported in GoLite. A thorough description of the language features of GoLite can be found at: https://www.cs.mcgill.ca/~cs520/2019/.

Our report will present the important design decisions in the implementation process of the compiler. In section 2, we describe the software technologies used and the reason why we have chosen them. We then break down the compiler into several parts and introduce each of them from section 3 to 8. All significant design decisions of each part of the compiler are also documented. Afterwards, we summarize our experience of implementing the compiler in section 9. Finally, in section 10, we record the work distribution of each member.

2 Language and Tool Choices

We chose C/flex/bison as our toolchain to implement the compiler, because these are the technologies that each group member is familiar with, which makes it easier for everyone to contribute to the project and saves us the time needed to learn new technologies. This has facilitated the development process and allowed us to complete the project within our time and efforts budget.

Our compiler converts an input program written in GoLite to an output program written in Python 3, which is one of the most popular programming languages nowadays. We decided to make Python 3 the target language because of the following advantages of Python:

- Since Python is dynamically typed, there is no need to handle blank identifiers specifically.
- Slice syntax, underline equality of objects and multiple assignments in GoLite are all natively implemented in Python.

We believe that these features of Python will reduce the workload of implementing the code generator.

3 Scanner

Scanning is the first phase of code compilation. A scanner performs lexical analysis on an input program. For each word passed into the scanner as an input, scanner checks if it is defined as a token in the flex file. It does so by trying to use every rule, defined with regular expressions, to match the input. The rule that has the longest first match will be applied to the input.

3.1 Design Decisions

3.1.1 Optional semicolons

Semicolons are optional in GoLite programs but are required in the formal grammar "as terminators in a number of productions"[2]. To support this, we added a global variable "NEED_SEMI_CONLON" to note that we need to return a semicolon before the next line. This variable is set to true according to the rules below, which are specified in the official go document, and it is set to false after inserting an semicolon in the tokens stream.

"A semicolon is automatically inserted after a line's final token if that token is

- an identifier
- an integer, floating-point, imaginary, rune, or string literal
- one of the keywords break, continue, fallthrough, or return
- one of the operators and punctuation ++, --,),], or }" [2].

3.1.2 Token naming

We used plain english instead of symbol semantics for all tokens' names, since at this stage we do not need to have deep knowledge of the language semantics. Doing so has saved us some time when implementing the scanner.

3.2 Testing

We wrote some test cases by referring to the language specification of GoLite and a test script to automatically execute the test cases to test our scanner. We categorized the test cases as valid input programs and invalid programs, and then check if the scanner outputs "OK" when being given a valid program and outputs an error message when being given an invalid program. At this stage, any programs that contain non-tokens are invalid programs.

4 Parser

Parsing is the second phase of code compilation. Only programs that have passed the scanning phase can reach the parsing phase. A parser performs syntactic analysis on an input program. It verifies that all the tokens in the input program are organized in a way such that a matching grammar rule in the bison file can be found for every line of code in the program.

4.1 Design Decisions

4.1 GLR algorithm

We used GLR algorithm with bison annotation %glr-parser to solve a reduce-reduce conflict [3][4], which is that an identifier list can be reduced to both expression list and identifier list. With LR(1), it is impossible to distinguish between these two. Therefore, we must add this annotation to the parser to resolve the conflict. This may impede the performance of the parser to a certain degree, but it shall not be significant for our use case.

4.2 Data types

We used "long" to store int values and "char *" for rune values and string values.

4.3 Abstract Syntax Tree

Our AST design follows closely to the official GoLang specification since it guarantees the correctness of the design and reduces our workload.

4.2 Testing

We wrote some test cases by referring to the language specification of GoLite and a test script to automatically execute the test cases to test our parser. We categorized the test cases as valid input programs and invalid programs, and then check if the scanner outputs "OK" when being given a valid program and outputs an error message when being given an invalid program. At this stage, any programs that contain syntactic errors are invalid programs.

5 Weeder

A program that passes the parsing phase will reach the weeder. A weeder adds additional logic to handle syntactic errors that are difficult to capture in the parser.

5.1 Design Decisions

5.1.1 Blank identifiers

We captured all the incorrect use of blank identifiers in the weeder. A blank identifier is a "_" string. It is seen as a normal identifier by the parser. However, in many cases, the use of blank identifiers are forbidden [5]. Since the parser sees a blank identifier as a normal identifier, the incorrect use of blank identifiers cannot be detected by the parser. We decided to handle this in the weeder because we wanted to keep the parsing rules minimal by not distinguishing blank identifiers from other identifiers.

5.1.2 Switch statement

We captured the erroneous case where a switch statement contains multiple default cases in the weeder. This is because for our parser, a switch case clause, either a default one or a non-default one, is considered to be the same thing. If we separate default case clauses from

non-default case clauses during parsing, the related parsing rules might become over complicated.

5.1.3 Functions

We checked whether or not a function have a terminating statement in the weeder. This is because the definition of a terminating function is very complicated, so checking function termination in the parser is very unrealistic. Please refer to section 7.1 for more details.

5.2 Testing

Once the scanner is completed, we begin to develop the test cases and the parser simultaneously. There are three major sources for our test cases: we wrote test cases after careful reading of the language specification document and the official Go Spec; we have also parsed the sample code from the official GoLang Tutorial; lastly, we have tested our parser on snippets of code from various open source Go repository and removed the unsupported features such as imports and class. Whenever a certain requirement sounds ambiguous, we check the test case on the online Go Playground interpreter. If the requirement sounds extremely ambiguous, we double-check the test case with the reference compiler found on the course's teaching server.

6 Symbol Table

Programs that pass the weeder will later reach the symbol table. Symbol tables allows the compiler to keep track of defined variables and their respective types. It is implemented by searching for variable declarations while traversing the abstract syntax tree built for the source program. For every valid variable declaration, the identifier of the variable and its associated type are added to the symbol table of the scope where the declaration takes place. Scopes are created and then exited according to the scoping rules defined in Section 6.8. Every scope has its own unique symbol table. Moreover, when implementing symbol tables, we also check if variable uses are legal, which is done by checking if every identifier used can be found in its current scope or any of its parent scopes.

We implemented symbol tables as a cactus stack of hash tables [6]. Each hash tables contains its parent and a set of symbols. Symbols (struct SYMBOL) are defined as structs and contains a string of the name of the symbol, a Type (struct TYPE) of the symbol, a kind (enum symbolKind) to tell apart of different kinds of symbols e.g type declaration, variable declaration, function declaration, and constants.

The important design decisions that we have made when implementing the symbol table is as follows.

6.1 Design Decisions

6.1.1 Variable Declarations: Initialization

For each variable declaration, we will first put the symbol of the expression into the symbol table. If there was not an explicit type notation, the type of the expression will be assigned to the variable. If there was not an expression, there must be an explicit type notation and this type information will be assigned to the variable. When declaring a variable with a type of slice or array, if the slice's/array's element type is declared, we will add this new slice/array type into symbol table.

6.1.2 Types

We added BaseTypeKind, underLineType, and infer kind to our existing TYPE struct definition. BaseTypeKind is an enumeration which is used when the type is a base type (int, float64, bool, string, rune). UnderLineType is a TYPE pointer which points to the underlying type. This creates a list of Types. The head of such a list is the current type pointer, while the tail is one of the base type pointers. For arrays and slices, its underLineType is the type of inner components. For example, the underLineType of a []int is int, the underLineType of a [][]int is []int. For structs, the id of its underLineType is always "struct" since we will not need to resolve to struct's inner components. We also changed how we set the identifier for slices and array types: the identifier of slices is "[]" plus the identifier of its underLineType, while the identifier of array is "[size]" plus the identifier of its underLineType. For instance, the identifier of a type for an int array with size being 5 is "[5]int". Additionally, at symbol table phase, if a variable does not have an explicit type notation, we will assign a type of kind k_infer and will assign the correct type information later.
When initializing the top level symbol table, we added all base types. To put a new type declaration symbol into symbol table, we first check if the name of the type is neither main nor init, and then we check if the type is recursive, and if so, we check if it is valid recursively. After all checks have been passed, we will only put the new type into symbol table if its identifier is not a blank identifier. We also set up the base type kind information for later use.

For struct type declarations, we will open a new scope for its field declarations. The identifier and type of each field declaration is added to the new symbol table.

We created a helper function to determine if two types are equal: bool checkSameType(TYPE *t1, TYPE *t2, bool checkBaseType). t1 and t2 are the two types to be checked. If checkBaseType is true, we check if two types' underlying base types are the same. Two types are equal if they have the same kind. For slice kind, two types are equal if they have the same element type. For array kind, two types are equal if they have the same element type and have the same size. For structs, two types are equal if all their field declarations have the same types and are declared in the same exact order, and the identifier of the structs are either identical or absent. We also check (in a separate function) the declaration scope of types. Two types are not equal if they are declared in different scopes.

6.1.3 Assign Statements

There are two checks for assign statements. First, the program validates that the left-hand side of the assign statement can be resolved to something that is resolvable (i.e. The lowermost layer of it must be referenceable by an identifier). Next, we check that both sides of the assign statement share the same type. During our development, we realized that two types sharing the same identifier could be considered as different types if they were declared in different scopes, and those types shall not be compatible in an assign statement or a binary expression. We therefore redesigned the procedure of filling our symbol table in a way that each unique type is pointed to by only one pointer.

6.1.4 Short Variable Declarations

As long as there is one undeclared variable on the left-hand side, a short variable declaration is valid. Since our identifier list implementation assumed that all elements of the list will share the same type, we made a few helper functions , such as resolveType() and strToType(), to ensure that only one internal copy of each unique type is stored in our table. Please note that a blank identifier cannot be the only undeclared variable.

6.1.5 Switch Statements

When a switch statement is detected, we first create a new symbol table A to store the new variable declared before we go into the switch cases. The symbol table of the scope where the switch statement as a whole occurs is the parent of symbol table A. Another symbol table, i.e. table B, needs to be created to store any possible new variable declarations in the switch case clauses. Since all the variables declared before the case clauses can be reached inside the case clauses, we must set table A to be the parent of table B. In addition, we must verify that any variables used in switch cases have been declared before.

6.1.6 For Loops

Our compiler merged the procedure of creating new scoped table for all block statement (statements enclosed in curly brackets) other than those found in switch, a function, or a struct. A new scope is created whenever a for statement, an if statement, or an explicit is detected. For example, for the for-loop, we first create a new symbol table A to store the new variable declared in the first part of the for clause. The symbol table of the scope where the for statement as a whole occurs is the parent of symbol table A. Another symbol table, i.e. table B, needs to be created to store any possible new variable declarations in the body of the for statement. Since all the variables declared before the for body can be reached inside the for body, we must set table A to be the parent of table B.

6.1.7 Functions

A new scope is created for afunction. The function's name and its parameters are declared at the beginning of this new table. Although our parser supported parameterized return variable, we abandoned it due to the complexity of implementation.

Since our symbol table has a one-pass implementation, in order to print the function arguments, we have created a global variable where the string representation of the function's signature is temporarily stored.

6.1.8 Scoping Rules

Any identifiers that have not been declared in a new scope may be used for variable declaration. They will shadow earlier declarations. Special cases include the parameter definition in a function declaration and the field definition in a struct declaration. In those cases, although declared types' names may be used as variables, their definitions as type will not be shadowed.

A new scope will be created in the following cases:

- In the body of for loop, if-else statements, or any other explicit block statement
- In the short statement of switch and the body of switch, please refer to Section 2.1.5
- In a function declaration, including its parameters
- In a struct declaration

A scope will be poped from our stack in the following case:

- When a closing curly brace is encountered (so that a block terminates).

6.2 Testing

We wrote some test cases by referring to the language specification of GoLite and a test script to automatically execute the test cases to test our parser. We categorized the test cases as valid input programs and invalid programs, and then check if the scanner outputs "OK" when being given a valid program and outputs an error message when being given an invalid program. At this stage, any programs that use not defined variables will be considered invalid.

7 Typechecker

After symbol tables are initialized, the input program reaches the typechecker. Type-checking allows the compiler to check if type compatibility is ensured in the source program. For every declaration, statement, or expression, whether it type-checks can be determined by verifying if respective rules defined in GoLite language specification[1] are satisfied.

For typechecker, our implementation closely follows the language specification, but there are some tricky cases that need extra work to handle. They are explained in Section 7.1.

7.1 Design Decisions

7.1.1 Function Typechecking and Return Statements

If a function is declared to return void, a return statement is not necessary, but if a return statement ever occurs in its body, we check that it returns a NULL expression.[3]

If a function is declared to return non-void type, first we must check that it always returns (i.e. It always terminates). This is implemented in weeder. We traverse the function body, and for each statement encountered, we check if it is a terminating statement. Repeat so until there are no more statements in the function body. When we reach the last statement of the function body, if it is not a terminating statement, then we can determine that this function does not terminate and an error must be thrown.

Terminating statements are defined as below:

- A return statement is a terminating statement
- A for statement which is equivalent to an infinite loop is a terminating statement
- An if statement is a terminating statement if its body terminates and its associated else statement is a terminating statement.
- An else statement that does not contain an if statement is a terminating statement if its body terminates
- An else statement that contains an if statement is a terminating statement if the contained if statement terminates
- A switch statement is a terminating statement if it contains a default case and every case ends with a terminating statement

After we make sure that a non-void function always returns, we check whether it returns what it claims to return when declared. We verify this by passing the return type of the function into statement type-checking function. In this way, when a return statement is detected while traversing the function body, we can check if the returned expression has the same type as the passed-in function return type.

7.2 Testing

We wrote some test cases by referring to the language specification of GoLite and a test script to automatically execute the test cases to test our parser. We categorized the test cases as valid input programs and invalid programs, and then check if the scanner outputs “OK” when being given a valid program and outputs an error message when being given an invalid program. At this stage, any programs that do not completely obey the typing rules of GoLite will be considered invalid.

8 Code Generator

The last phase of our compiler is the code generator. The code generator converts an input program in GoLite to a semantically equivalent program in Python 3.

8.1 Design Decisions

8.1.1 Scoping

The major difference between Go/GoLite and Python is the scoping. Unlike Go/GoLite, Python does not have block scoping. Instead, in the context of a single script file with no class definition involved, only functions will create a new scope in Python.
Naturally, one might attempt to simulate Go's/GoLite's scoping with Python's function scopes, but the problem is actually more complex. Consider the following code snippet:

```
1  package main
2
3  var a = 10
4
5  func printa(){
6          println(a)
7  }
8
9  func main() {
10         println(a)
11         a = 100
12         println(a)
13         var a = "go"
14         println(a)
15         printa()
16 }
17
```

Figure 1 Input program

```
10
100
go
100

Program exited.
```

Figure 2 Output

In Go/GoLite, two variables of different scopes sharing the same identifier may be used in the same block. However, Python mandates that an identifier in a particular function block must resolve to a particular variable of a predetermined scope.

Although it is possible to simulate Go's/GoLite's behavior by tracking variable usage and inject a new function scope in Python whenever it is needed, it sounds like an expensive solution. Instead, we decided to assign a unique identification number (UID) to every variable when we first put them into the symbol table, and rename all variables of the GoLite program to a new name that carries the said UID. In our implementation, each variable will then have an unique identifier, which flattens the scoping hierarchy of Python and allows Python to behave in the expected way.

Another potential issue shown by the above code snippet is that a GoLite program may attempt to re-assign the value of a variable of the outer scope. This is not allowed in Python

unless the variable has been explicitly declared as a global or a non-local[1] variable. Fortunately, GoLite does not support nested function declarations, meaning that issues may arise only when the program is trying to re-assign a global variable inside a function. We solve this issue by encapsulating all access to global variables with Python's "global()" built-in function.

8.1.2 Declarations

When a GoLite variable is declared, it is often initialized to a default value. Although Python does not require a variable to be explicitly declared before assignment, we translate a GoLite's variable declaration to an assignment to the variables' default values in Python code. Since Python supports multiple assignments, the initialization of multiple variables in one line shall not cause any issue.

Default values of variables of different data typs:
- Int: 0
- Float64: 0.0
- Boolean : False
- Rune: 0
- String: ""
- Slice: an empty Slice object (see section 8.1.10)
- Array: a list each element initialized to their default value
- Struct: a dictionary with key-value pairs initialized to their default value

Similarly, for type declarations, whenever a type is declared in GoLite, we simply assign the default value of the type to an identifier representing the type. Whenever a new variable is declared to have a particular type, we simply copy the value of the variable using the identifier to the new variable.

Please see section 8.1.4 for function declaration.

8.1.3 Assignments

GoLite's assignment statements are very similar to those in Python. Both of them support multiple assignments. Primitive values are passed by values in both languages. Since we are implementing arrays and structs using list() and dict() of Python, we have to call deepcopy() whenever those constructs are assigned to. Slices, on the other hand, are passed by the header pointer in GoLite, so it shall behave similarly to the Python class we implemented in terms of assignments.

[1] A non-local variable in Python is a variable defined outside of the current scope but not inside the global scope.

8.1.4 Functions

Functions in GoLite can be more or less directly translated to Python. There are only three main distinctions:

1. If a function attempts to modify an outer scope variable, issues may arise. See section 8.1.1 for more details.
2. If multiple blank identifiers are found in the function's signature, a direct translation will not be accepted in Python. We opt to replace those blank identifiers with temporary variables of the form "___{int}" to resolve this issue.
3. Although GoLite defines a function's return type, this is not necessary in Python. In fact, the value returned by a Python function does not have to be consistent. Therefore, we do not have to handle the special case where an infinite loop acts as a terminating statement.

8.1.5 Loops

For statements in GoLang are used to execute a block repeatedly. There are three types of for statements supported by GoLite: infinite, while, and 3-part style. We will start with the 3-part style for loop. This style of for loop consists of an init statement, a condition, and a post statement. Both the init statement and the post statement can be defined using a simple statement. The init statement may be a short variable declaration, while the post statement may not. Before the for-loop is executed for the first time, the init statement will be executed. Afterwards, in each iteration, GoLite will check whether the condition is satisfied. If it is satisfied, the body of the for loop will be executed. Once the execution of the body is finished, the post statement will be executed. Note that if the condition is not satisfied, or if the loop is terminated by break statement, the for loop will exit without executing the post statement.

For a 3-part for loop, any part of the clause may be empty, but the two semicolons must be present. In the case where there is only a condition statement in the clause, the two semicolons may be omitted. This style of for loop is called "while loop style" in GoLite. The procedure in executing this is very similar to the while loop in many other languages. In the case where the condition is also empty, a special kind of infinite loop is defined. Both "for {}" and "for ;; {}" formats are accepted. Of course, infinite loops can also be implemented by using loops with a condition that always evaluates to the constant "true", i.e. "for true{}". However, this does not fall under infinite loop style according to GoLang's compiler. Unlike the infinite loops with empty statements, which are considered as terminating if no break statement is found inside the body, the while-loop style infinite loop cannot be used as a terminating statement in neither GoLang or GoLite. Fortunately, Python does not require functions to declare their return types in their signatures. Functions in Python also do not have to return a consistent type. Therefore, we will assume that all programs that pass our

typechecker have functions with valid terminating statements and we do not have to write any special processes for the special infinite loop.

Turning to mapping strategies, Python's for loops share more similarities to for-range loops in Go, which is not part of GoLite. Instead, we opt to use while-loops to implement GoLite's for loops. Although Python's loops do not come with a new scope, our implementation of UID mentioned in section 8.1.1 should handle this properly. The for loop's init statement will go to the first line inside the function, and the condition will be declared as the while loop's condition, and the post statement will be placed as the last line inside the translated while loop's body. In addition, the post statement is also added before any "continue" statements.

8.1.6 If statement

If statements are used to control the execution of different branches based on the evaluation result of boolean expressions associated with the branches. In GoLite, a simple statement is allowed immediately after the "if" keyword, and it will be executed before the if condition is evaluated. An if branch gets executed if and only if the corresponding condition evaluates to true. Zero or more elif statements can follow an if statement. An else if branch gets executed if and only if every if/elif branch before it has a false condition and its own condition evaluates to true. An else statement can come immediately after an if statement or if/elif statements. The else block does not get executed unless every if/else if branch before it has a false condition.

Since every if statement in GoLite may contain a simple statement, we will not use Python's "elif" construct. Instead, every else-if branch will be split into two. If there is a simple statement following the "if" keyword in the input program, when performing the mapping, we need to extract it and put it before each if statement. In the case where a simple statement follows an "else if" keyword, we will decompose it into an "else" branch and an inner "if" statement, then insert the simple statement into the else branch.

8.1.7 Switch statement

Go and Python are on two different end of spectrum when it comes to switch statement. Go's switch is very powerful, it can even evaluate values on the fly, whereas Python does not support switch statements at all. Therefore, we will use Python's if-elif-else constructs to simulate Go's/GoLites's switches. Before the first if statement, we shall first translate GoLite switch's simple statement. After that, we shall assign the result of GoLite switch condition to a temporary variable, so the expression will be evaluated only once. Each clause following GoLite's case keyword is translated to an if or an elif branch, whereas the default case is translated to an else or an "if True" branch, depending on whether it is the only case in the switch. The translation is fairly straightforward since Go's cases break by default. Considering that the fallthrough functionality is not supported by GoLite, the behavior can be accurately simulated by if statements.

Since GoLite uses lazy evaluation when it evaluates each case, if there exist multiple conditions in one case, we connect those condition with an “or” keyword in Python.
In order to handle break statements inside the switch, we have created a new Exception class and simulate the break statement behavior using the class. To keep things consistent and simple, we also use the Exception class for break statements found in while-loops.

8.1.8 Structs

We used python’s built-in dictionary to represent structs, where the stringified field name are used as key. Since GoLite structs cannot be printed, we opt to use the built-in dictionary instead of the ordered data structure for better performance. We leave the rest, such as underline equality check, to Python’s native implementation.

8.1.9 Arrays

We used Python’s built-in lists to represent arrays. Since GoLite’s arrays cannot be dynamically resized, we do not have to care about the underlying implementation of GoLite. Python’s lists shall be able to accurately simulate GoLite’s array behavior.

8.1.10 Slices

We have created a custom Python class to simulate slices’ behavior.A slice in GoLite is represented by a header pointer that points to an array storing the slice’s values and a length attribute that indicates the last accessible element in the slice. Whenever append() is called, the program simply tries to modify the element that is at an index identical to the length attribute. If more capacity is needed, the program will create a new array while doubling the capacity. Most of those are easy to translate. However, there is one issue. Without creating multiple layers of list() that will certainly hinder the performance, we need to find an alternative way to save the different “length” attributes shared by slices whose header pointer point to the same underlying structure. Since we have already created a UID for every identifier, we decided to store the corresponding length of each UID in a dictionary attribute of the class.

However, since we have only implemented UID for identifiers, some valid expressions, such as elements of arrays, structs, or function return values, may not be able to pass a usable UID. Considering our time budget, instead of revamping our symbol table, we handle the problem with fault-masking techniques. If a method cannot decide whether an operation is valid or not based on UIDs, the method will always assume the operation is valid and perform an additional check at a later stage.

8.1.11 Prints

The print statements in GoLite only support the printing of simple types that can be casted to a basic type. Among those basic types, two need special handling: boolean and float. Boolean

in Python are capitalized, whereas the opposite is true for GoLite. Float is a bit more complicated. By default, GoLite prints floats in a scientific format with 6 trailing significant figures and 3 exponent digits. This differs from Python's scientific format with only 2 exponent digits printed and the third one printed only if required. We have to pass the arguments of the print function to a filter that stringifies the values according to GoLite's rule.

8.1.12 Miscellaneous

There exist equivalent expressions in Python for most operators in GoLite. Only the bitclear operator is missing in Python, but it is equivalent to the combination of bitwise "and not" operator.
In addition, to handle casting expressions, we created a helper function that first checks the type of the expressions in question and call the appropriate Python functions.

8.2 Testing

Before starting the development of this phase in full-force, we have read through the GoLite Tutorial slides and performed numerous tests on the Go Playground to understand the possible edge cases surrounding Go's semantics. Some of those tests forced us to re-design parts of our code generator. Once the first prototype was completed, we moved on to the debugging phase.

Due to the time constraint during this phase of development, we had to focus our testing on things that were most likely to go wrong. Since many constructs of GoLite can be converted directly to some Python constructs, we trusted that our design shall perform as expected. Thus, the great majority of our time was spent on testing the "slice" construct from GoLite, which behaves counter-intuitively in many scenarios. We spent the remaining time testing on "switch", another point of differentiation of GoLite. Some other constructs, such as "struct", were completely skipped in our tests.

9 Conclusion

Our experience of implementing this compiler is both bitter and sweet. We have devoted a lot of personal time into the project. Sometimes we even had to sacrifice the time that should have been used to study for several exams to work on it. However, we still had a lot of fun throughout the implementation process. We remember the joy and sense of achievement we had every time we successfully implemented a new feature. We enjoy making design decisions to achieve our goal eventually. At the end of the project, we are truly amazed at what an interesting language Go/GoLite is, and we have earned a lot of technical knowledge in return. All of us are able to appreciate the art of programming more than ever before. More importantly, what we have learnt has enabled us to view programming from a different perspective, which can potentially help us become better software engineers. Last but not

least, we have developed invaluable friendship with each other while working together in many days and nights.

If we had the opportunity to go back in time, there are several things we would like to change. For the non-technical part, we should have looked at the course policy extremely carefully before working on the project, so that we could be clearly aware of the necessity of citations and the forbidden use of grading materials.

Turning to technical aspect, we would like to change how we handle optional semicolons in the scanner. Instead of manually setting a variable to denote if we need to insert a semicolon, we could have stored the last returned token and then returned semicolons according to the rules. We also wish that we spent more time on designing the symbol table's structure and properly decoupling it from the typechecker. Since the beginning of the development, we always felt reluctant to refactor our symbol table implementation. We always had to fix issues caused by the symbol table by adding more values and more properties to it. Many of those could have been avoided had we revamped it earlier. Additionally, although our compiler passes all typechecking test cases, there probably still exist quite a few hidden issues that we did not have the time to find out.

10 Contributions

We collaborated with one another during development by using Visual Studio Code with VS Live Share extension. In this way, everyone can access, modify, and execute the source code at the same time, which has highly improved our development efficiency. At the end of each collaboration session, one of the team members committed the changes to GitHub for version control.

All team members have made significant contributions to the implementation of each part of the compiler as well as each milestone report.

11 References

[1] Shiotsu, Y. (2018). "Golang: A General Purpose Programming Language Made for the Future" [Online] upwork.com. Available at: https://www.upwork.com/hiring/development/golang-programming-language/

[2] Google. (2018). "The Go Programming Language Specification" [Online] golang.org. Available at: https://golang.org/ref/spec#Semicolons

[3] Athena GNU Locker. (2000). "The Bison Parser ALgorithm" [Online] web.mit.edu. Available at: http://web.mit.edu/gnu/doc/html/bison_8.html

[4] Dodd, C. (2011). "C++ GLR parsers with Bison" [Online] stackoverflow.com. Available at: https://stackoverflow.com/questions/8526887/c-glr-parsers-with-bison

[5] Krolik, A. (2019). "COMP520 - Blank Identifier Specification" [Online] cs.mcgill.ca. Available at: https://www.cs.mcgill.ca/~cs520/2019/project/Blank_Specifications.pdf.

[6] Krolik, A. (2019). Course slides 7-symbol.pdf. [Online] cs.mcgill.ca. Available at: https://www.cs.mcgill.ca/~cs520/2019/slides/7-symbol.pdf.